Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 19:05*
Temat *Algorytmy populacyjne*	**Problem** *TSP*
Skład grupy *241281 Karol Kulawiec*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *9 kwietnia 2020*

1 Opis problemu

Algorytmy populacyjne, w porównaniu do algorytmów lokalnego przeszukiwania, przechowują całą grupę rozwiązań i na ich podstawie buduje się kolejną grupę rozwiązań, które zastąpią bieżącą. Algorytmy populacyjne inspirują się procesami ewolucji naturalnej oraz odkryciami z dziedziny genetyki.
Zbadany algorytm populacyjny: Algorytm genetyczny.

2 Algorytm genetyczny

Algorytm genetyczny (Genetic Algorithm) to algorytm heurystyczny, przeszukujący przestrzeń alternatywnych rozwiązań problemu w celu znalezienia rozwiązań najlepszych. Wykorzystuje mechanizm dostosowywania się żywych organizmów do otaczającego ich środowiska na drodze ewolucji. Algorytm spośród swojej populacji, poprzez określoną selekcję, wybiera rozwiązania, które następnie krzyżuje ze sobą i tworzy nowe osobniki. Jak to w naturze bywa, nowym osobnikom zdarzają się różne wynaturzenia, tak i w algorytmie uwzględnia się mutację osobników.

W algorytmie zbadano wpływ:

- selekcji: ruletka, turniejowa, rankingowa, losowa;
- operatora krzyżowania: jednopunktowy, OX, PMX;
- operatora mutacji: swap, insert, invert;
- rozwiązania hybrydowego.

Główny algorytm selekcyjny, w zależnie od wybranej metody selekcyjnej (wybor_selekcji), oraz wg ustawionych parametrów tej selekcji, dokonuje selekcji. Algorytm prezentuje się w następujący sposób:

Listing 1: selekcja_calosciowa()

```
1  def selekcja_calosciowa(populacja, wybor_selekcji=3, precyzja_ruletki=2,
       rozmiar_turnieju=2, metoda_selekcji_rankingowej=1,uwzglednienie_plci=
       False,wielkosc_populacji=100): #wybor i zmiana wszystkich par
2      populacja_arr = np.array(populacja)
3      ilosc_par = int(len(populacja)/2)
4      pary = []
5      if wybor_selekcji==0: #Losowa:
6          if uwzglednienie_plci:
7              a = np.random.randint(0, int(wielkosc_populacji/2), size=[
                   ilosc_par,1])
8              b = np.random.randint(int(wielkosc_populacji/2),
                   wielkosc_populacji, size=[ilosc_par,1])
9              pary = np.append(a,b, axis=1)
10         else:
11             pary = np.random.randint(0, wielkosc_populacji, size=[ilosc_par
                   ,2])
12
13     elif wybor_selekcji==1: #ruletka
14         nr_pary = 0
15         while nr_pary < ilosc_par:
16
17             if precyzja_ruletki == 0: #bez zwiekszenia precji selekcyjnej
```

```
18                  f_celu = np.array(populacja)
19                  f_celu = f_celu[:,0]
20                  rodzice = wybranie_rodzicow(f_celu)
21
22              elif precyzja_ruletki == 1: #zwiekszenie presji selekcyjnej (
                    zwiekszenie preferencji wybierania lepszych rozwiazan)
23                  przerobione_f_celu = []
24                  stare_f_celu = np.array(populacja)
25                  stare_f_celu = stare_f_celu[:,0]
26                  max_ze_starych = np.max(stare_f_celu)
27                  for stara_f_celu in stare_f_celu:
28                      nowa_f_celu = stara_f_celu/max_ze_starych
29                      przerobione_f_celu.append(nowa_f_celu)
30                  rodzice = wybranie_rodzicow(przerobione_f_celu)
31
32              elif precyzja_ruletki == 2: #wieksze zwiekszenie presji
                    selekcyjnej (moze byc za duze)
33                  przerobione_f_celu = []
34                  stare_dlugosci = np.array(populacja)
35                  stare_dlugosci = stare_dlugosci[:,1]
36                  min_z_dlugosci = np.min(stare_dlugosci)
37                  for stara_dlugosc in stare_dlugosci:
38                      nowa_dlugosc = stara_dlugosc - min_z_dlugosci + 1
39                      przerobione_f_celu.append(1/nowa_dlugosc)
40                  rodzice = wybranie_rodzicow(przerobione_f_celu)
41
42              pary.append(rodzice)
43              nr_pary += 1
44
45      elif wybor_selekcji==2: #turniejowa
46          nr_pary = 0
47          while nr_pary < ilosc_par:
48              rodzice = [0,0]
49              for i in range(2):
50                  if uwzglednienie_plci:
51                      uczestnicy_turnieju = np.random.randint(int(
                            wielkosc_populacji/2)*i, int(wielkosc_populacji/(2-i
                            )), size=[1,rozmiar_turnieju])
52                  else:
53                      uczestnicy_turnieju = np.random.randint(0,
                            wielkosc_populacji, size=[1,rozmiar_turnieju])
54                  rozmiary = []
55                  for j in range(rozmiar_turnieju):
56                      rozmiary.append(populacja_arr[uczestnicy_turnieju[0,j
                            ],1])
57                  min_rozmiar = np.min(rozmiary)
58                  wygrany_rodzic = 0
59                  for j in range(rozmiar_turnieju):
60                      if rozmiary[j] == min_rozmiar:
61                          wygrany_rodzic = uczestnicy_turnieju[0, j]
```

```
62                          break
63                  rodzice[i] = wygrany_rodzic
64
65              pary.append(rodzice)
66              nr_pary += 1
67
68      elif wybor_selekcji==3: #rankingowa
69          prawdopodobienstwa = []
70          rangi = np.array([], dtype=int)
71          arr_argsort = np.argsort(populacja_arr[:,1])
72          for i in range(len(populacja)):
73              rangi = np.insert(rangi,0,i+1)
74          if metoda_selekcji_rankingowej == 0: # liniowa
75              for i in range(len(populacja)):
76                  wspolczynnik_selekcji = 2.0 #[1.0, 2.0]
77                  funkcja_R = 2-wspolczynnik_selekcji+(2*(
                        wspolczynnik_selekcji-1)*(rangi[i]-1))/(len(populacja)
                        -1)
78                  prawdopodobienstwa.append(funkcja_R)
79          elif metoda_selekcji_rankingowej == 1: # nieliniowa
80              wspolczynnik_selekcji = 1.41 #dobrany eksperymentalnie
81              sum_wsp_selekcji = suma_wsp_selekcji(wspolczynnik_selekcji, len
                    (populacja))
82              for i in range(len(populacja)):
83                  funkcja_R = (len(populacja)*np.power(wspolczynnik_selekcji,
                         rangi[i]-1)/(sum_wsp_selekcji))
84                  prawdopodobienstwa.append(funkcja_R)
85
86          #Normalizacja funkcji R:
87          prawdopodobienstwa = normalizacja_r(prawdopodobienstwa)
88          nr_pary = 0
89          while nr_pary < ilosc_par: #Teraz podobnie jak w ruletce
90
91              losowe_liczby = np.random.rand(2)
92              rodzice = [0, 0]
93              for nr_liczby, losowa_liczba in enumerate(losowe_liczby):
94                  suma_prawd = 0
95                  if uwzglednienie_plci:
96                      prawdopodobienstwa = np.array(prawdopodobienstwa)*2
97                      if nr_liczby==1:
98                          losowa_liczba += 1
99                  for num, prawdopodobienstwo in enumerate(prawdopodobienstwa
                        ):
100                     suma_prawd += prawdopodobienstwo
101                     if (num == len(prawdopodobienstwa)-1) or (losowa_liczba
                            < suma_prawd):
102                         rodzice[nr_liczby] = arr_argsort[num]
103                         break
104             pary.append(rodzice)
105             nr_pary += 1
```

```
106
107     return pary
```

Funkcje pomocnicze:

Listing 2: wybranie_rodzicow()

```
1  def wybranie_rodzicow(f_celu, uwzglednienie_plci=False):
2      suma_f_celu = np.sum(f_celu)
3      losowe_liczby = np.random.rand(2)
4      rodzice = [0, 0]
5      for nr_liczby, losowa_liczba in enumerate(losowe_liczby):
6          prawdopodobienstwa = []
7          for cel in f_celu:
8              prawdopodobienstwa.append(cel/suma_f_celu)
9          suma_prawd = 0
10
11         if uwzglednienie_plci:
12             prawdopodobienstwa = np.array(prawdopodobienstwa)*2
13             if nr_liczby==1:
14                 losowa_liczba += 1
15
16         for num, prawdopodobienstwo in enumerate(prawdopodobienstwa):
17             suma_prawd += prawdopodobienstwo
18             if (num == len(prawdopodobienstwa)-1) or (losowa_liczba <
                   suma_prawd):
19                 rodzice[nr_liczby] = num
20                 break
21
22     return rodzice
```

Listing 3: normalizacja_r()

```
1  def normalizacja_r(prawdopodobienstwo):
2      nowe_prawdopodobienstwo = []
3      suma = 0
4      for prawd in prawdopodobienstwo:
5          suma += prawd
6      for prawd in prawdopodobienstwo:
7          nowe_prawdopodobienstwo.append(prawd/suma)
8      return nowe_prawdopodobienstwo
```

Listing 4: suma_wsp_selekcji()

```
1  def suma_wsp_selekcji(wspolczynnik_selekcji, dlugosc):
2      suma = 0
3      for i in range(dlugosc):
4          suma += np.power(wspolczynnik_selekcji,i)
5      return suma
```

Algorytm krzyżowania wygląda następująco:

Listing 5: krzyzowanie()

```
1  def krzyzowanie(para, operator=2):
2      rodzic1 = para[0]
3      rodzic2 = para[1]
4
5      if operator == 0: # jednopunktowa - pierwsza polowa taka sama, druga to
           dopisanie liczb z drugiego rodzica
6          miejsce_podzialu=int(len(rodzic1)/2)
7          rodzic1_half = rodzic1[:miejsce_podzialu]
8          rodzic1_copy = rodzic1.copy()
9          rodzic2_half = rodzic2[:miejsce_podzialu]
10         rodzic2_copy = rodzic2.copy()
11
12         for item in rodzic2_half:
13             rodzic1_copy = rodzic1_copy[rodzic1_copy!=item]
14         for item in rodzic1_half:
15             rodzic2_copy = rodzic2_copy[rodzic2_copy!=item]
16
17         dziecko1 = np.array(list(rodzic2_half) + list(rodzic1_copy))
18         dziecko2 = np.array(list(rodzic1_half) + list(rodzic2_copy))
19
20     if operator == 1: #OX
21         k1, k2 = np.random.randint(0, len(rodzic1), size=[1,2])[0]
22         if k1>k2:
23             k1,k2=k2,k1
24
25         dziecko1 = rodzic1[k1:k2+1]
26         dziecko2 = rodzic2[k1:k2+1]
27         n_r_2 = np.array(list(rodzic2[k2+1:])+list(rodzic2[:k2+1]))
28         n_r_1 = np.array(list(rodzic1[k2+1:])+list(rodzic1[:k2+1]))
29
30         for value in n_r_2:
31             if value not in dziecko1:
32                 dziecko1 = np.append(dziecko1, value )
33         dziecko1 = np.append(dziecko1[len(dziecko1)-k1:], dziecko1[:len(
               dziecko1)-k1])
34
35         for value in n_r_1:
36             if value not in dziecko2:
37                 dziecko2 = np.append(dziecko2, value )
38         dziecko2 = np.append(dziecko2[len(dziecko2)-k1:], dziecko2[:len(
               dziecko2)-k1])
39
40     if operator == 2: #PMX
41         k1, k2 = np.random.randint(0, len(rodzic1), size=[1,2])[0]
42         if k1>k2:
43             k1,k2=k2,k1
44
45         dziecko1 = np.zeros(len(rodzic1))
46         dziecko2 = np.zeros(len(rodzic1))
```

```
47
48          przedzial1=rodzic1[k1:k2+1]
49          przedzial2=rodzic2[k1:k2+1]
50
51          dziecko1[k1:k2+1] = przedzial1
52          dziecko2[k1:k2+1] = rodzic2[k1:k2+1]
53
54          dziecko1 = daj_dziecko(dziecko1, dziecko2, rodzic1,rodzic2,
                przedzial1, przedzial2)
55          dziecko2 = daj_dziecko(dziecko2, dziecko1, rodzic2,rodzic1,
                przedzial2, przedzial1)
56
57      return dziecko1, dziecko2
```

Funkcje pomocnicze:

Listing 6: daj_dziecko()

```
1  def daj_dziecko(dziecko1, dziecko2, rodzic1, rodzic2, przedzial1,
       przedzial2):
2      for item in przedzial2:
3          if item not in przedzial1:
4              miejsce = rekurencyjne_szukanie(rodzic1, rodzic2, item,
                   przedzial1, przedzial2)
5              dziecko1[miejsce] = item
6
7      for i in range(dziecko1.size):
8          if dziecko1[i]==0:
9              dziecko1[i] = rodzic2[i]
10     return dziecko1
```

Listing 7: rekurencyjne_szukanie()

```
1  def rekurencyjne_szukanie(rodzic1, rodzic2, item, przedzial1, przedzial2):
2      index = np.argwhere(np.array(rodzic2)==item)[0,0]
3      item2 = rodzic1[index]
4      if item2 in przedzial2:
5          miejsce = rekurencyjne_szukanie(rodzic1, rodzic2, item2, przedzial1
               , przedzial2)
6      else:
7          miejsce = np.argwhere(np.array(rodzic2)==item2)[0,0]
8      return miejsce
```

Algorytm mutacyjny:

Listing 8: mutacja()

```
1  def mutacja(path, wariant=2, rozmiar=10):
2      a,b = zaburzonePozycje(rozmiar)
3
4      if wariant==0: #SWAP
5          path = swap_sciezka(path,a,b)
6
```

```
7       if wariant==1:
8           path = insert_sciezka(path,a,b,rozmiar)
9
10      if wariant==2:
11          path = invert_sciezka(path,a,b)
12
13      return path
```

Funkcje swap_sciezka(), inseert_sciezka() oraz invert_sciezka() nie różnią się od metod z poprzedniego etapu.

W rozwiązaniu hybrydowym zastosowano algorytm Local Search 2OPT, który wykonuje się do momentu znalezienia pierwszego polepszenia.

Listing 9: ls_2opt()

```
1  def ls_2opt(arr,populacja, wersja_ls, jedna_iteracja, rozmiar):
2      #nowe_populacje = populacja.copy()
3      if wersja_ls==0:
4          nowe_populacje = ls_greedy(arr,populacja, jedna_iteracja=
               jedna_iteracja, rozmiar=rozmiar) #first improvement, first
               descent
5      if wersja_ls==1:
6          nowe_populacje = ls_steepest(arr,populacja, jedna_iteracja=
               jedna_iteracja, rozmiar=rozmiar) #best impovement, highest
               descent
7
8      return nowe_populacje
```

Algorytm może wykonać się albo zachłannie, albo ostro:

Listing 10: ls_greedy()

```
1  def ls_greedy(arr,populacje, jedna_iteracja=True, rozmiar=10):
2      nowe_populacje = populacje.copy()
3      for nr, populacja in enumerate(populacje):
4          path = populacja[2]
5          length = populacja[1]
6          while True:
7              best_new_path = []
8              best_new_length = length
9              nowy=False
10             for i in range(len(path)-1):
11                 for j in range(i+1, len(path)):
12                     new_path = path.copy()
13                     new_path[i],new_path[j]=new_path[j],new_path[i]
14                     new_length = new_length = countPathFaster(length, path,
                           new_path, arr, rozmiar,i,j)
15                     #new_length = countPath(new_path, arr, rozmiar)
16                     if new_length < best_new_length:
17                         best_new_length = new_length
18                         best_new_path = new_path
19                         nowy=True
20                         break
```

```
                if nowy:
                    break
            if best_new_length < length:
                length = best_new_length
                path = best_new_path
                if jedna_iteracja:
                    break
            else:
                break

        nowe_populacje[nr][2] = path
        nowe_populacje[nr][1] = length
        nowe_populacje[nr][0] = 1/length
    return nowe_populacje
```

Listing 11: ls_steepest()

```
def ls_steepest(arr,populacje, jedna_iteracja=True, rozmiar=10):
    nowe_populacje = populacje.copy()
    for nr, populacja in enumerate(populacje):
        path = populacja[2]
        length = populacja[1]
        while True:
            best_new_path = []
            best_new_length = length
            for i in range(len(path)-1):
                for j in range(i+1, len(path)):
                    new_path = path.copy()
                    new_path[i],new_path[j]=new_path[j],new_path[i]
                    new_length = countPathFaster(length, path, new_path,
                        arr, rozmiar,i,j)
                    #new_length = countPath(new_path, arr, rozmiar)
                    if new_length < best_new_length:
                        best_new_length = new_length
                        best_new_path = new_path

            if best_new_length < length:
                length = best_new_length
                path = best_new_path
                if jedna_iteracja:
                    break
            else:
                break

        nowe_populacje[nr][2] = path
        nowe_populacje[nr][1] = length
        nowe_populacje[nr][0] = 1/length
    return nowe_populacje
```

Dodatkowo, w celu przyśpieszenia obliczeń długości ścieżki, zaimplementowano algorytm, wykonujący stałą liczbę operacji:

Listing 12: countPathFaster()

```
def countPathFaster(old_length, old_path, path, arr, rozmiar,i,j):
    length = old_length
    if i==0:
        length -= arr[0][old_path[0]]
        length -= arr[old_path[0]][old_path[1]]

        length += arr[0][path[0]]
        length += arr[path[0]][path[1]]
    else:
        length -= arr[old_path[i-1]][old_path[i]]
        length -= arr[old_path[i]][old_path[i+1]]

        length += arr[path[i-1]][path[i]]
        length += arr[path[i]][path[i+1]]

    if j== rozmiar-2:
        length -= arr[old_path[j-1]][old_path[j]]
        length -= arr[old_path[j]][old_path[0]]

        length += arr[path[j-1]][path[j]]
        length += arr[path[j]][path[0]]
    else:
        length -= arr[old_path[j-1]][old_path[j]]
        length -= arr[old_path[j]][old_path[j+1]]

        length += arr[path[j-1]][path[j]]
        length += arr[path[j]][path[j+1]]


    return length
```

2.1 Algorytm i jego przebieg

W celu porównania algorytmu do algorytmów przeszukiwań lokalnych, warunkiem końcowym jest ten sam czas trwania algorytmu, który nie przekracza 7s.

W algorytmie, w celu zmniejszenia możliwych kombinacji, przyjmujemy elitaryzm, wielkość populacji równą 100, prawdopodobieństwo krzyżowania równe 90%, prawdopodobieństwo mutacji równe 10%, mutowanie przy pomocy metody invert.

Algorytm najpierw w losowy sposób wybiera populację początkową, następnie zapisywany jest elitarny osobnik, który zostanie dodany do listy, po wykonaniu wszystkich operacji. W głównej pętli, wybierane przy pomocy selekcji są pary (tutaj 50 par, ponieważ 100 osobników w populacji), a następnie dla każdej pary, przy pomocy krzyżowania i mutowania, tworzy się ich dzieci. Dzieci dodane zostają do nowej populacji, która po wykonaniu wszystkich dzieci, zastąpi bieżącą populację. Z wszystkich dzieci, sprawdzane jest, czy któreś z nich jest nowym najlepszym rozwiązaniem, elitarny osobnik zastępuje najgorszego na liście osobników, następnie wybierany jest nowy elitarny, z bieżącej populacji. Na koniec, jeżeli algorytm uwzględnia wersję hybrydową, dla każdego dziecka, wywołuje się algorytm Local Search.

Listing 13: Algorytm genetyczny

```
def obliczenieCzasuWyniku(arr, rozmiar,czy_hybrydowy,wersja_ls,
    czy_jedna_iteracja_w_ls,czy_rozroznienie_plci,nr_selekcji,
    precyzja_ruletki,ilosc_uczestnikow_turnieju,metoda_selekcji_rankingowej,
    metoda_krzyzowania):
    elitaryzm=True
    wielkosc_populacji = 100
    nr_pokolenia = 1
    p_k = 0.9 # Prawdopodobienstwo krzyzowania, 0.6-1.0
    p_m = 0.1 # Prawdopodobienstwo mutacji, 0.01 - 0.1
    max_czas = 7.0 # maksymalny czas dzialania algorytmu
    wariant_mutowania = 2
    populacja_poczatkowa = [] #funckja celu = 1/dlugosc, dlugosc, sciezka
    for i in range(wielkosc_populacji):
        path = np.random.permutation(range(1,rozmiar))
        length = countPath(path, arr, rozmiar)
        populacja_poczatkowa.append([1/length,length,path])

    najkrotsza_dlugosc, najlepsze_rozwiazanie = najlepsze_z(
        populacja_poczatkowa)


    start = datetime.datetime.now()
    duration = datetime.datetime.now() - start
    populacja = populacja_poczatkowa

    arr_pop=np.array(populacja)
    elitarni_osobnicy=[]
    if elitaryzm:
        elitarny = populacja_poczatkowa[np.argmin(arr_pop[:,1], axis=0)]

    while duration.total_seconds() < max_czas:
        # wybranie wielkosc_populacji/2 par
        pary = selekcja_calosciowa(populacja, wybor_selekcji=nr_selekcji,
            precyzja_ruletki=precyzja_ruletki, rozmiar_turnieju=
            ilosc_uczestnikow_turnieju, metoda_selekcji_rankingowej=1,
            uwzglednienie_plci=czy_rozroznienie_plci,wielkosc_populacji=
            wielkosc_populacji)
        j = 0
        k = int(len(populacja_poczatkowa)/2)
        populacja_nowa = []
        while j<k:
            dziecko1 = populacja[pary[j][0]][2]
            dziecko2 = populacja[pary[j][1]][2]
            dzieci = []
            dzieci.append(dziecko1)
            dzieci.append(dziecko2)
            R = np.random.rand(3)
            if R[0] < p_k:
                dziecko1, dziecko2 = krzyzowanie(dzieci, operator=
```

```
                metoda_krzyzowania)
42              if R[1] < p_m:
43                  dziecko1 = mutacja(dziecko1, wariant=wariant_mutowania,
                        rozmiar=rozmiar)
44              if R[2] < p_m:
45                  dziecko2 = mutacja(dziecko2, wariant=wariant_mutowania,
                        rozmiar=rozmiar)
46
47              dziecko1 = dziecko1.astype(int)
48              dziecko2 = dziecko2.astype(int)
49              length1 = countPath(dziecko1, arr, rozmiar)
50              length2 = countPath(dziecko2, arr, rozmiar)
51              populacja_nowa.append([1/length1,length1,dziecko1])
52              populacja_nowa.append([1/length2,length2,dziecko2])
53
54              j += 1
55
56          populacja = populacja_nowa
57          najkrotsza_dlugosc_z_obecnej_populacji,
                najlepsze_rozwiazanie_z_obecnej_populacji = najlepsze_z(
                populacja)
58          if najkrotsza_dlugosc_z_obecnej_populacji < najkrotsza_dlugosc:
59              najkrotsza_dlugosc = najkrotsza_dlugosc_z_obecnej_populacji
60              najlepsze_rozwiazanie =
                    najlepsze_rozwiazanie_z_obecnej_populacji
61
62          arr_pop=np.array(populacja)
63          if elitaryzm:
64              najgorszy = np.argmax(arr_pop[:,1], axis=0)
65              if populacja[najgorszy][1] > elitarny[1]:
66                  populacja[najgorszy] = elitarny
67
68              arr_pop=np.array(populacja)
69              elitarny = populacja[np.argmin(arr_pop[:,1], axis=0)]
70
71          if czy_hybrydowy:
72              populacja = ls_2opt(arr,populacja, wersja_ls=wersja_ls,
                    jedna_iteracja=czy_jedna_iteracja_w_ls, rozmiar=rozmiar)
73
74          duration = datetime.datetime.now() - start
75
76      return najkrotsza_dlugosc
```

3 Eksperymenty obliczeniowe

Obliczenia, tak samo jak poprzednio, zastały wykonane na laptopie z procesorem i5-7300HQ, kartą graficzną NVIDIA GeForce GTX 1050, 16GB RAM i DYSK SSD oraz zostały napisane w języku Python.

Zebrano 216 danych. Z powodu tak dużej liczby danych, wszystkie dane nie zostaną zaprezentowane, a jedy-

nie to, co udało się na ich podstawie wywnioskować. Porównywane będą jakości wyników, które obliczane były poprzez podzielenie optymalnego rozwiązania, przez uzyskany wynik. Ponieważ uzyskany wynik, nie mógłby być mniejszy niż optymalny, wyniki mieszczą się w zakresie (0.0 - 1.0], gdzie 1.0 oznacza uzyskanie optymalnego wyniku, 0.5 uzyskanie dwukrotnie gorszego wyniku, itp.

3.1 Wnioski

Porównanie jakości wyników wg następujących parametrów:

Tablica 1: Rozwiązanie hybrydowe

Czy rozwiązanie hybrydowe	min	mean	max
Tak	0.228	0.544	1.0
Nie	0.235	0.519	1.0

Tablica 2: Wersja rozwiązania hybrydowego

Wersja LS	min	mean	max
Greedy	0.240	0.583	1.0
Steepest	0.228	0.505	1.0

Tablica 3: Wersja selekcji

Wersja selekcji	min	mean	max
Losowa	0.228	0.498	1.0
Ruletka	0.228	0.525	1.0
Turniejowa	0.230	0.551	1.0
Rankingowa	0.232	0.557	1.0

Tablica 4: Presja najlepszych osobników w ruletce

Presja	min	mean	max
Zwykła	0.228	0.515	1.0
Zwiększona	0.228	0.516	1.0
Mocno zwiększona	0.234	0.542	1.0

Tablica 5: Ilość uczestników turnieju

Ilość	min	mean	max
2	0.231	0.549	1.0
10	0.230	0.553	1.0

Tablica 6: Metoda selekcji rankingowej

Metoda	min	mean	max
liniowa	0.233	0.556	1.0
nieliniowa	0.232	0.558	1.0

Tablica 7: Metoda krzyżowania

Krzyżowanie	min	mean	max
jednopunktowe	0.229	0.545	1.0
OX	0.228	0.531	1.0
PMX	0.228	0.532	1.0

Z powyższych danych wynika, że dla:

- rozwiązania hybrydowego:
  lepiej wypadło zastosowanie rozwiązania hybrydowego, który mimo spowolnienia działania algorytmu, korzystnie wpłynął na wynik,
- wersji rozwiązania hybrydowego:
  lepiej wypadła wersja greedy, wg mnie spowodowane jest to tym faktem, że ponieważ i tak akceptujemy pierwszą poprawę wyniku, algorytm greedy szybciej znajduje pierwszy lepszy wynik, niż algorytm steepest przeszuka całe sąsiedztwo bieżącego wyniku, co korzystnie wpływa na ilość wykonanych iteracji algorytmu,
- wersji selekcji:
  najlepszą wersją okazała się wersja rankingowa, następnie turniejowa, a następnie ruletka,
- presji najlepszego osobnika w ruletce: najlepsze wyniki uzyskano dla największej presji najlepszego osobnika,
- ilości uczestników turnieju:
  lepiej wypadła większa grupa,
- metody selekcji rankingowej:
  z małą przewagą lepiej wypadła metoda nieliniowa,
- metody krzyżowania:
  co zadziwiające, najlepiej sprawdziło się jednopunktowe, najmniej skomplikowane rozwiązanie.

Najlepsze uzyskane wyniki dla poszczególnych rozmiarów:

Tablica 8: Najlepsze uzyskane wyniki

rozmiar	jakość wyniku	wynik	optymalny wynik
21	1.000	2707	2707
71	0.668	2917	1950
323	0.305	4347	1326

Jak widać, algorytm dla małych wartości, potrafi dać optymalny wynik. Dla dużych, uzyskany wynik w tak krótkim czasie, różni się znacznie od optymalnego. Porównując czasy, do czasów algorytmu SA oraz TS, algorytm GA mieści się pomiędzy nimi.

Kombinacje parametrów, które dały 5 najlepszych wyników:

Tablica 9: TOP 5 kombinacji parametrów dla rozmiaru 71

czy hybrydowy	wersja ls	selekcja	specyfikacja selekcji	metoda krzyżowania	jakość wyniku
True	greedy	losowa	brak	jednopunktowe	0.668495
True	greedy	ruletka	zwiększona presja	jednopunktowe	0.634146
True	greedy	ruletka	zwykła presja	jednopunktowe	0.574374
True	greedy	turniej	2 uczestników	jednopunktowe	0.537042
True	greedy	rankingowa	nieliniowy	jednopunktowe	0.532496

Tablica 10: TOP 5 kombinacji parametrów dla rozmiaru 323

czy hybrydowy	wersja ls	selekcja	specyfikacja selekcji	metoda krzyżowania	jakość wyniku
True	greedy	rankingowa	liniowa	OX	0.305038
True	greedy	turniej	10 uczestników	OX	0.302119
True	greedy	rankingowa	nieliniowa	OX	0.293948
True	greedy	rankingowa	nieliniowa	PMX	0.293039
True	greedy	turniej	10 uczestników	PMX	0.291621

Jak widać w zestawieniu, nie ma jednej kombinacji wyników, która dała by najlepsze wyniki dla każdego rozmiaru. Widać, że opłaca się zastosować hybrydowe rozwiązanie, w wersji greedy, jednak wersję selekcji i krzyżowania należy dobrać eksperymentalnie.

4 Wnioski

Algorytm genetyczny to bogate narzędzie z duża możliwością rozwoju. Na w pełni przetestowanie wszystkich konfiguracji algorytmu, zabrakło czasu. Być może, algorytm gorzej wypadł od algorytmu SA, ponieważ badanie odbywało się przez 7s i algorytm genetyczny nie mógł w pełni wykorzystać swojej przewagi polegającej na sprawniejszym wychodzeniu z optimum lokalnych.